Polytech Dijon

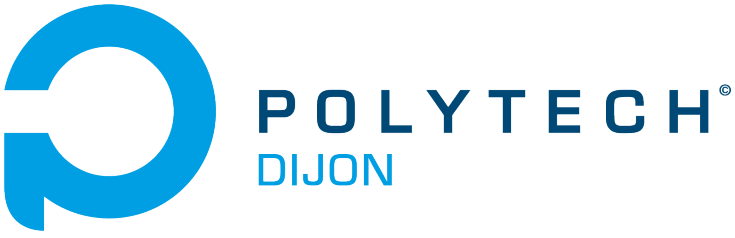
FISA IOT 5A

Compte rendu projet VHDL

Contrôleur SRAM

Auteur :
HELLE Evan

Enseignant :
DUBOIS Julien



2025-2026

Sommaire

Table des Figures

Ce document a été réalisé avec LaTeX.

1 Compte Rendu

1.1 Introduction

Ce projet consiste à concevoir et implémenter en VHDL un contrôleur pour une mémoire SRAM de type mt55l512y36f, dont le modèle VHDL est fourni par le fabricant.
L'objectif est de permettre les opérations de lecture et d'écriture tout en respectant les contraintes temporelles et les spécifications électriques du composant.

Le contrôleur recevra les données via une interface d'entrée-sortie (IO Buffer) et assurera la communication avec la mémoire SRAM à travers une interface dédiée.
Son fonctionnement sera structuré autour d'une machine à états finis (FSM) afin de gérer de manière séquentielle et fiable les différentes phases d'accès mémoire.

Le projet inclura également la réalisation de `test bench` destinés à valider le comportement et la conformité de chaque élément du contrôleur.

1.2 IO Buffer

1.2.1 Fonctionnement

Un IO Buffer est un composant matériel utilisé dans les circuits pour gérer les signaux d'entrée et de sortie entre le circuit et le monde extérieur.
L'implémentation de L'IO Buffer que nous allons utilisé nous nous a été fourni.

Voici la représentation schématique de l'IO Buffer que nous allons utilisé dans notre projet :

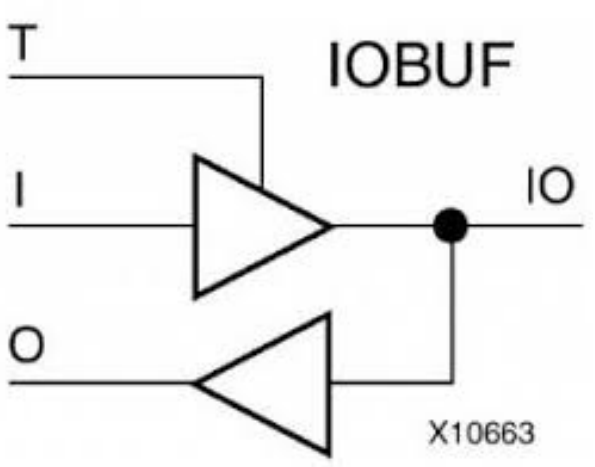


Figure 1.1: IO buffer

L'IO Buffer est composé de plusieurs entrées et sorties :

- **I** : Donnée d'entrée (input data)
- **O** : Donnée de sortie (output data)
- **T** : Trigger, permet de contrôler le mode de l'IO Buffer (entrée ou sortie qui servira à gerer l'ecriture ou la lecture dans notre cas)
- **IO** : Bus de données bidirectionnel (data bus)

1.2.2 Test Bench

L'objectif de faire un test bench est de vérifier le bon fonctionnement de l'IO Buffer avec la SRAM.
De plus cela permettra de de comprendre son fonctionnement avant de l'intégrer dans le projet final.
Il est important de faire ce test car il est nécessaire d'avoir un décalage entre la donnée et les signaux de contrôle au niveau de la SRAM.

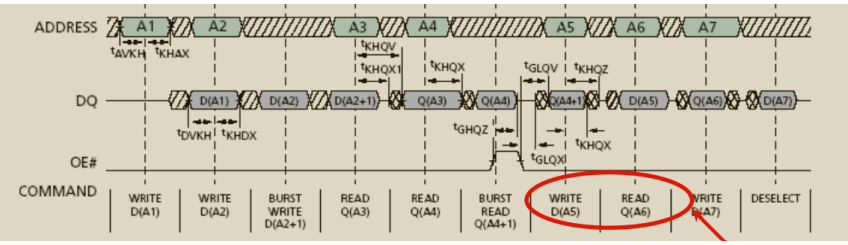


Figure 1.2: Extrait énoncé projet

D'après l'énoncé du projet et la datasheet de la SRAM, il est nécessaire d'avoir un décalage d'un cycle d'horloge entre la donnée et les signaux de contrôle.

Code

Pour le code du test bench, nous sommes partie sur une ecriture de donnée à l'adresse 1 et 2, puis une lecture de ces deux adresses.

```
tb : PROCESS
BEGIN
-- init
nCKE    <= '0';
nADVLD  <= '0';
nOE     <= '0';-- output enable
nCE     <= '0';
nCE2    <= '0';
CE2     <= '1';
SA      <= (others => '0');
Trig    <='1'; -- se mettre en "lecture" le temps de l'init pour ne pas ecrire n'importe quoi

wait for 6*(TCLKL+TCLKH);
```

```
SA                  <= "000"&x"0001";
Trig    <= '0'; -- ecriture à l'adresse 1
wait for 1*(TCLKL); -- pour travailler sur front montant
ENTREE  <= (others => '1'); -- decalage de la donnée d'un cycle
                            --par rapport à l'adresse et la commande


wait for 2*(TCLKL+TCLKH);
SA                  <= "000"&x"0002"; -- eriture à l'adress 2
wait for 1*(TCLKL+TCLKH);
ENTREE  <= ENTREE + 1;-- decalage de la donnée d'un cycle par
                      -- rapport à l'adresse et la commande

wait for 2*(TCLKH+TCLKL);
SA                  <= "000"&x"0001"; -- lecture à l'adresse 1
Trig <= '1';
wait for 2*(TCLKH+TCLKL);
SA                  <= "000"&x"0002"; -- lecture à l'adresse 2

wait; -- will wait forever
END PROCESS;
```

Ici la donnée est bien décalée d'un cycle par rapport à l'adresse et la commande. Quand le trigger est à 0, on est en écriture, et quand il est à 1, on est en lecture. C'est le cas pour la SRAM et l'IO buffer. Donc nous avons connecter par un fil ces deux I/O.

```
SRAM1 : mt551512y36f port map
(DQ, SA, '0', CLKO_SRAM, nCKE, nADVLD, '0',
'0', '0', '0', Trig, nOE, nCE, nCE2, CE2, '0');

IOb: for I in 0 to 35 generate
Iobx: IOBUF_F_16  port map(
O => SORTIE(I),
IO => DQ(I),
I => ENTREE(I),
T => Trig
);
end generate;
```

Simulation

Voici le résultat de la simulation :

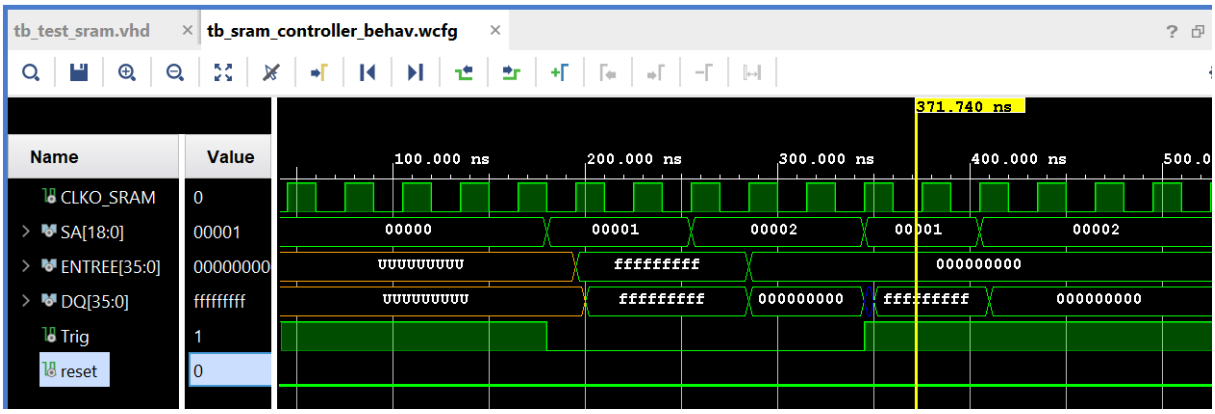


Figure 1.3: Simulation IO Buffer

Les données sont bien écrites aux bonnes adresses (1 et 2) et lues correctement par la suite.
Au passage à la lecture, le bus bidirectionnel passe en haute impédance, ce qui est le comportement attendu.
Voici un zoom sur le passage en haute impédance :

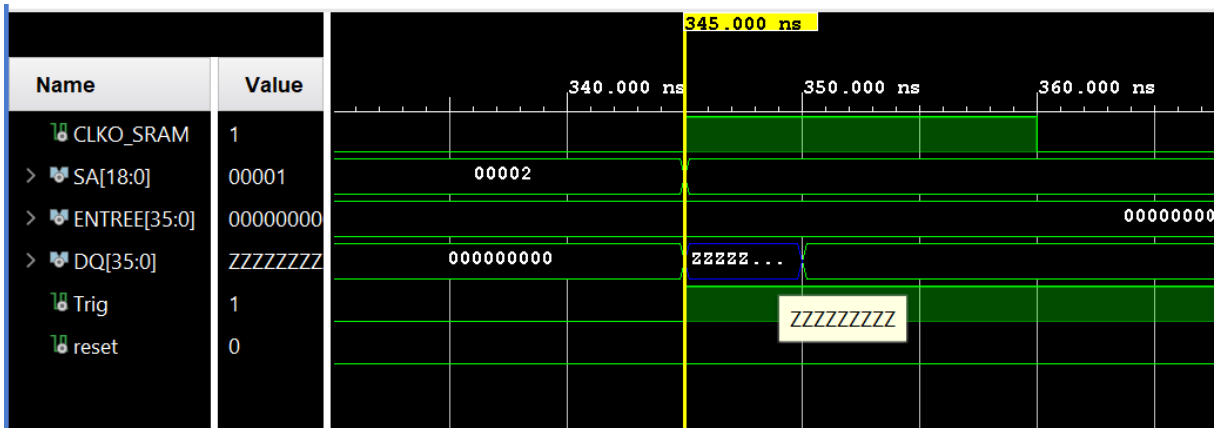


Figure 1.4: Haute impédance à la lecture de la donnée

1.3 Contrôleur SRAM

Maintenant que nous avons validé le bon fonctionnement de l'IO buffer connecté à la SRAM, nous allons implémenter le contrôleur de la SRAM. Celui-ci sera responsable de la gestion des opérations de lecture et d'écriture vers la mémoire SRAM en fonction des signaux d'entrée.
Son fonctionnement sera basé sur une machine à états finis (FSM) qui gérera les différents états nécessaires pour effectuer les opérations de lecture et d'écriture.
un IO buffer sera également présent pour interfacer les données entre le contrôleur et la SRAM.

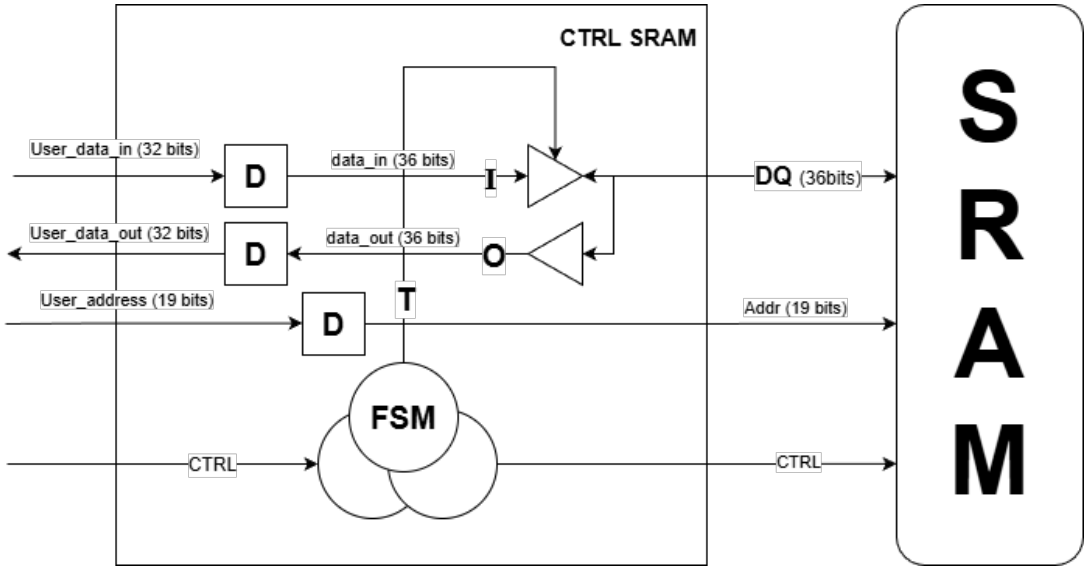


Figure 1.5: Contrôleur SRAM

Coté utilisateur, Les signaux ont été ajustés pour tensir sur un nombre d'octets complet:

- `User_address` : Adresse de 19 bits
- `User_data_in` : Données d'entrée de 32 bits
- `User_data_out` : Données de sortie de 32 bits
- `Ctrl` : Signal d'écriture/lecture (1 bit)

Nous avons fait le choix de laisser l'adresse sur 19 bits pour pouvoir adresser toute la mémoire SRAM.
D'un point de vue utilisateur, nous aurions pu choisir de mettre l'addresse sur 16 bits pour avoir un nombrecomplet d'octet.

1.3.1 Finished State Machine

D'après la page 11 de la datasheet de la SRAM :

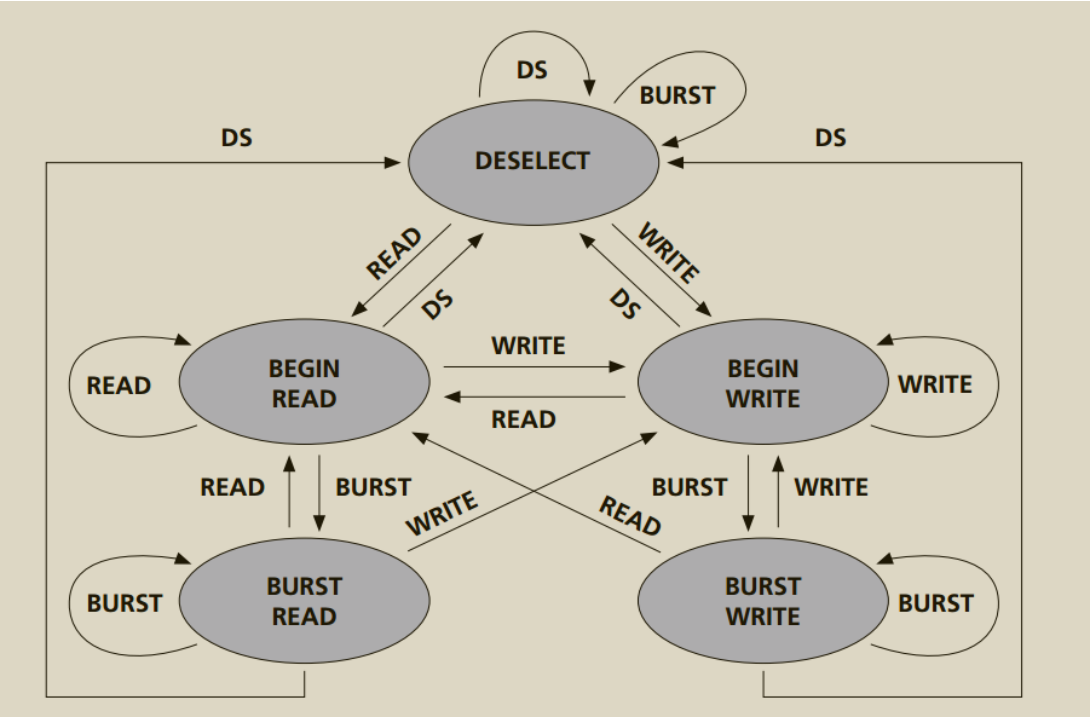


Figure 1.6: FSM

Pour le moment, nous n'allons pas implémenter le mode Burst.
Cela signifie que les signaux `MODE (LBO#)` et `ADV/LV#` seront à 0.

Implementation FSM

```
process(Clk,reset)
begin
if reset ='1' then
    state  <= INIT;

elsif Clk'EVENT and Clk = '1' then
    case state is
        When INIT =>
            state  <= IDLE;

        when IDLE =>
            if Ctrl ='1' AND Start ='1' then
                state <= READ;
            elsif Ctrl ='0' AND Start ='1' then
                state <= WRITE;
            else
                state <= IDLE;
            end if;

        when READ =>
            if Ctrl ='1' AND Start ='1' then
                state <= READ;
            elsif Ctrl ='0' AND Start ='1' then
                state <= WRITE;
```

```
            else
                state <= IDLE;
            end if;

        when WRITE =>
            if Ctrl ='1' AND Start ='1' then
                state <= READ;
            elsif Ctrl ='0' AND Start ='1' then
                state <= WRITE;
            else
                state <= IDLE;
            end if;
    end case;
end if;
end process;
```

1.3.2 Implementation du Contrôleur SRAM

D'après le sujet, plusieurs signaux seront à mettre à 0 :

- ZZ sooze mode : basse consommation. 1 pour activer
- nCKE : clock enable si mis à 0
- nBWA
- nBWB
- nBWC
- nBWD : Byte write enable. permet de selectionner quels octets sont écrits. Mis à 0 pour écrire tous les octets.

Ces signaux seront mis à 0 dans un état précédent l'état `IDLE` qui est l'état `INIT`.

Afin de creer un déclalge d'horloge entre la donnée utilisateur, l'adresse et la commande, nous allons utiliser des bascules D La manière la plus simple d'utiliser des bascules D est de assaigner une entrée à un signal interne du composant dans un processus sensible aux fronts de l'horloge.

Ici, sur front monant, nous allons décaler la donnée utilisateur de deux cycles d'horloge et recopier l'adresse utilisateur sur l'adresse de la SRAM.
Sur front descendant, nous allons ajouter les bits de parité à la donnée avant de l'envoyer à la SRAM et renvoyer la donnée lue de la SRAM vers la sortie utilisateur.

```
process(Clk)
begin
if Clk'EVENT and Clk = '1' then
--decalage de la donnée de deux fronts montant
decalage_data_in_1 <= User_Data_in ;
decalage_data_in_2 <= decalage_data_in_1;
```

```
7
8  Addr <= User_Address; -- recopier l'adresse d'entrée du controller en entrée de la SRAM
9  end if;
10 end process;
11
12 process(Clk)
13 begin
14 if Clk'EVENT and Clk = '0' then -- sur front descendant
15 Data_in_s <= "0000" & decalage_data_in_2; --Ajout des 4 bits de parités dans à la donnée
16 User_Data_out <= Data_out_s(31 downto 0); --Renvoie des 32 bits sur la sortie
17 end if;
18 end process;
```

1.3.3 Test bench

Le test bench de notre composant SRAM complet se résume à tester plusieurs points:

- Que les instructions de lecture/ecriture données par l'utilisateur son bien traitées par la FSM.
- Les données à écrire/lire sont bien traitées par l'IO Buffer.
- Les données sont correctements transmisent à la SRAM.

pour se faire, nous allons d'abord executer une sequence d'écriture/lecture à l'adresse 0. Ensuite, nous feront la même chose à l'adresse 7FFF... (soit la dernière adresse de la SRAM). Et enfin pour vérifier que la SRAM stock bien les données, nous vérifirons que la donnée à l'adresse 0 est toujours présente

Ces tests nous permettront de nous assurer du bon fonctionnement de notre implémentation du Contrôleur SRAM.

Le test bench est disponile sur GitHub sous le nom de fichier `tb_sram_controller.vhd`.

Voici sur la page suivante, la simulation du test bench:
La configuration de la simulation est disponible dans le répertoire GitHub sur le nom de `tb_sram_controller_behav.wcfg`.

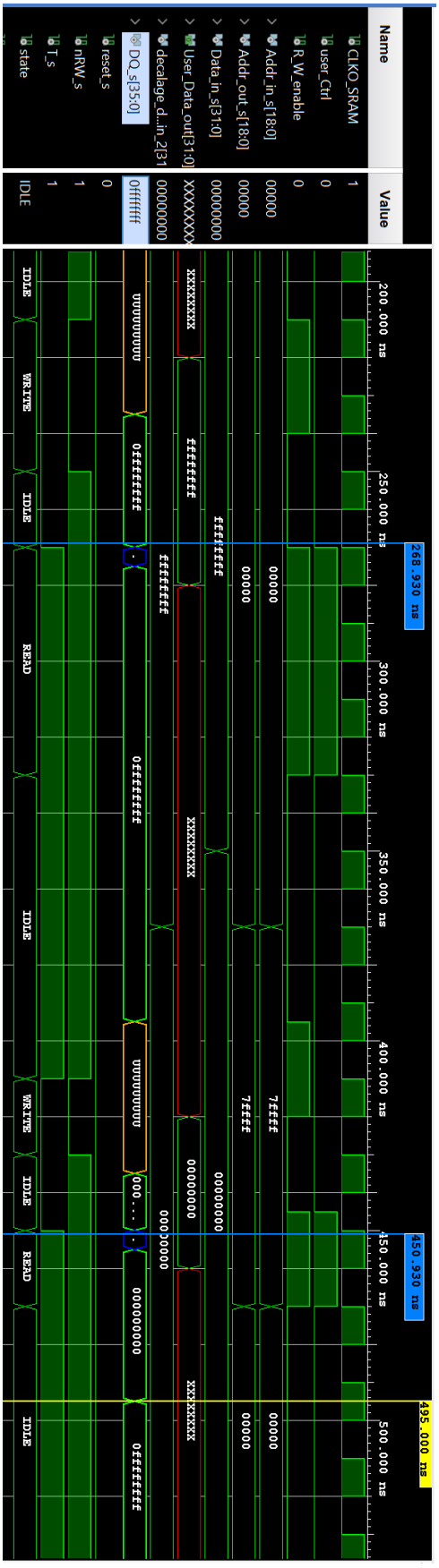


Figure 1.7: Simulation Test Bench Contrôleur SRAM

Des marqueurs ont été placés pour indiquer les lectures aux différentes adresses. On constate que les données lues sont bien celles qui ont été écrites précédemment. à la lecture, le bus DQ de la SRAM est mis en haute impédance, ce qui est visible sur la simulation.

Les données et les adresses sont bien décalées d'un cycle d'horloge, comme prévu.

Ainsi, nous avons validé le bon fonctionnement de notre Contrôleur SRAM.

1.4 Mode Burst

La SRAM est capable de fonctionner en mode Burst. Ce mode permet de lire ou d'écrire plusieurs mots consécutifs en une seule opération, ce qui améliore les performances lors de l'accès à des blocs de données.
Nous allons implémenter ce mode dans notre contrôleur SRAM pour optimiser les opérations de lecture et d'écriture.

1.4.1 Implémentation du mode Burst

Premièrement, le signal `ADV/LV#0` varira en fonction du mode de fonctionnement.
Pour l'instant, il est à l'état de constante à 0.
Nous allons donc assigner cette variable à 0 dans notre FSM à l'état `IDLE`, `READ` et `WRITE`.

Après cette étape, un rapide lancement de la simulation du Contrôleur SRAM indique que ce changement n'a rien cassé.

FSM

Pour l'implémentation du mode Burst, nous allons ajouter deux nouveaux états à notre FSM existante : `READ_BURST` et `WRITE_BURST`.
Ces états géreront les opérations de lecture et d'écriture en mode Burst.

De plus il faut désormais gérer des signaux que nous avions laissé de côté précédemment :

- Ld_n, permet d'autoriser ou de bloquer le changement d'adresse lors d'une opération en cours.
- CE_n, nécessaire à 0 si Ld_n est à 0 pour que l'adresse soit prise en compte.
- OE_n,

Voici un extrait de la FSM avec la gestion de ces signaux:

```
process(state)
    begin
        case state is
            When INIT =>
                ...
            when IDLE =>
                Rw_n <= '1'; --protection materiel
                CE_n   <= '1';
                OE_n   <= '1';
                Ld_n   <= '1';
                T_s    <= '0';

            when READ =>
                Rw_n <= '1'; -- on assignera cette valeur au trigg de la SRAM par la suite
                T_s  <='1';
                CE_n   <= '0';
                OE_n   <= '0';
                Ld_n   <= '0'; -- BEGIN
                --Addr   <= User_Address;

            when WRITE =>
                Rw_n <= '0';
                T_s  <='0';
                CE_n   <= '0';
                OE_n   <= '1';
                Ld_n   <= '0'; -- BEGIN
                --Addr   <= User_Address;

            when READ_BURST =>
                ...
            when WRITE_BURST =>
                ...
        end case;
    end process;
```

En ajoutant la gestion de ces signaux, Il faut s'assurer que le mode de fonctionnement hors Burst reste intact.
Nous allons donc relancer une simulation pour vérifier que tout fonctionne correctement. Le test bench à d'ailleur été amélioré pour afficher la dernière lecture de données sur l'adresse 0.

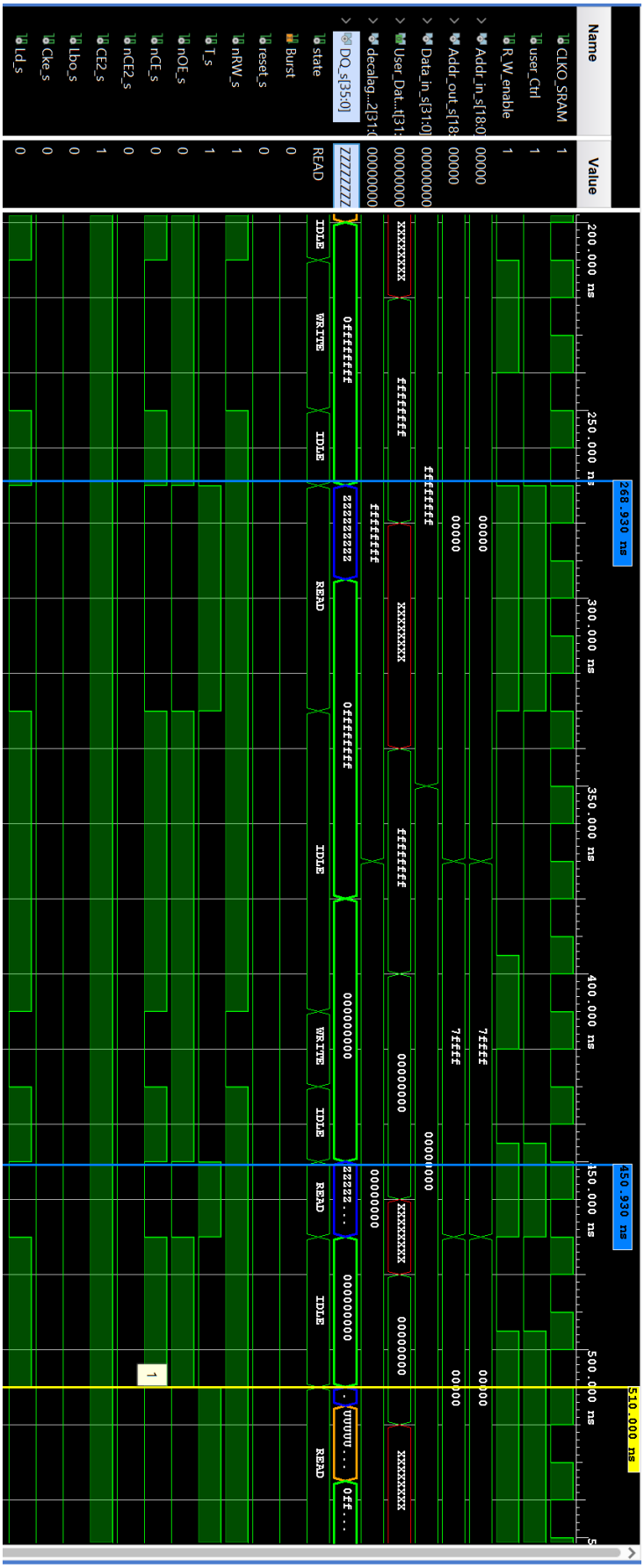


Figure 1.8: Simulation après ajout des signaux CE_n, OE_n et Ld_n

La logique de changement d'état de la FSM doit également être mise à jour pour inclure les nouveaux états de Burst.
Voici un extrait de la logique de changement d'état mise à jour :

```
1
2  process(Clk,reset)
3      begin
4          if reset ='1' then
5              state  <= INIT;
6
7          elsif Clk'EVENT and Clk = '1' then
8              case state is
9                  When INIT =>
10                     state  <= IDLE;
11
12                 when IDLE =>
13                     if Ctrl ='1' AND Start ='1' AND Burst ='0' then
14                         state <= READ;
15                     elsif Ctrl ='0' AND Start ='1' AND Burst ='0' then
16                         state <= WRITE;
17                     else
18                         state <= IDLE;
19                     end if;
20
21                     when READ =>
22                         if Burst = '1' AND Start = '1' then
23                             state <= READ_BURST;
24                         elsif Ctrl = '1' AND Start = '1' then
25                             state <= READ;
26                         elsif Ctrl = '0' AND Start = '1' then
27                             state <= WRITE;
28                         else
29                             state <= IDLE;
30                         end if;
31
32                     when WRITE =>
33                         if Burst = '1' AND Start = '1' then
34                             state <= WRITE_BURST;
35                         elsif Ctrl = '1' AND Start = '1' then
36                             state <= READ;
37                         elsif Ctrl = '0' AND Start = '1' then
38                             state <= WRITE;
39                         else
40                             state <= IDLE;
41                         end if;
42
43                 when READ_BURST =>
44                     if Ctrl ='1' AND Start='1' AND Burst ='0' then
45                         state <= READ;
46                     elsif Ctrl ='0' AND Start='1' AND Burst ='0' then
47                         state <= WRITE;
48                     elsif Burst ='1' AND Start='1' then
49                         state <= READ_BURST;
50                     else
```

```
                        state <= IDLE;
                    end if;

                when WRITE_BURST =>
                    if Ctrl ='1' AND Start='1' AND Burst ='0' then
                        state <= READ;
                    elsif Ctrl ='0' AND Start='1' AND Burst ='0' then
                        state <= WRITE;
                    elsif Burst ='1' AND Start='1' then
                        state <= WRITE_BURST;
                    else
                        state <= IDLE;
                    end if;

            end case;
        end if;
    end process;

```

DQ est passé en bidirectionnel inout ce n'était pas le cas avant nous utilisionss un signal qui décalait la donnée pour la mettre en sortie ou en entrée.

Nous avons été contraint d'ajouter un signal intermédiaire T_s_reg qui gère la direction du bus de données bidirectionnel.
Si la direction du bus change instantanément sans passer par un registre, nous risquons un conflit de bus : le FPGA et la SRAM essaient de parler en même temps sur le même fil pendant une fraction de seconde, ce qui crée des courts-circuits en simulation.
T_s_reg permet de s'assurer que l'ordre de passer en haute impédance est parfaitement aligné avec le cycle d'horloge où la donnée doit sortir.

Nopus avons également ajouté un registre pour l'adresse de burst : burst_addr.
Il faut séparer clairement adresse de départ et adresse courante du burst.

Il faut incrementer la donnée sur le write burst pour voir si tout va bien.
Une simple boucle dans le test bench permet de faire cela.

Il y'a eu une confusion avec le signal Ld_n.
Nous avons forcé l'incrementation de l'adresse par le Contrôleur durant le mode burst.
Or, la SRAM est capable de gérer l'incrementation automatique de l'adresse en mode burst si Ld_n est à 1.
Si nous voulons faire un programme plus preformant, il faudrait reprendre le code et changer l'incrementation forcée par celle de la sram.
Par souci de temps, nous laisserons cela ainsi pour le moment. Cela est amplement suffisant pour valider le fonctionnement du Contrôleur.

Voici la partie du test bench pour le mode burst:
Nous avons fait le choix d'ajouter cette partie au test bench existant plutot que de créer un nouveau test bench dédié au mode burst.

```
-- MODE BURST TB ------------------------------------------------------
```

```
        wait for 1*(TCLKH+TCLKL);
        Data_in_s <= "00000000000000000000000000000001"; -- 32 bits

    wait for 1.25*(TCLKH+TCLKL);
    Addr_in_s <= "00000000000000000001"; -- 19 bits
    wait for 1.25*(TCLKH+TCLKL); --write
        user_Burst <='0';
        user_Ctrl <= '0';
        R_W_enable <='1';
        --nRW_s    <= '0';

    wait for 1.25*(TCLKH+TCLKL); --write BURST
        user_Burst <='1';
        -- Boucle d'incrémentation manuelle synchronisée sur l'horloge
    for i in 0 to 4 loop
        wait until rising_edge(CLKO_SRAM);
        Data_in_s <= std_logic_vector(unsigned(Data_in_s) + 1);
    end loop;

        wait for 8.75*(TCLKH+TCLKL);--read
        user_Burst <='0';
        --wait for 1.25*(TCLKH+TCLKL);
        Addr_in_s <= "00000000000000000001"; -- 19 bits on va relire les données écrit avec le bur
        user_Ctrl <= '1';
        R_W_enable <='1';
        --nRW_s    <= '1';

        wait for 2.5*(TCLKH+TCLKL);--read BURST
        user_Burst <='1';
        user_Ctrl <= '1';
        R_W_enable <='1';
        --nRW_s    <= '1';
```

L'écriture en mode burst fonctionne parfaitement, de même pour la lecture.
nous incrémentation la donnée à chaque cycle d'horloge pendant le burst pour vérifier que tout fonctionne correctement.

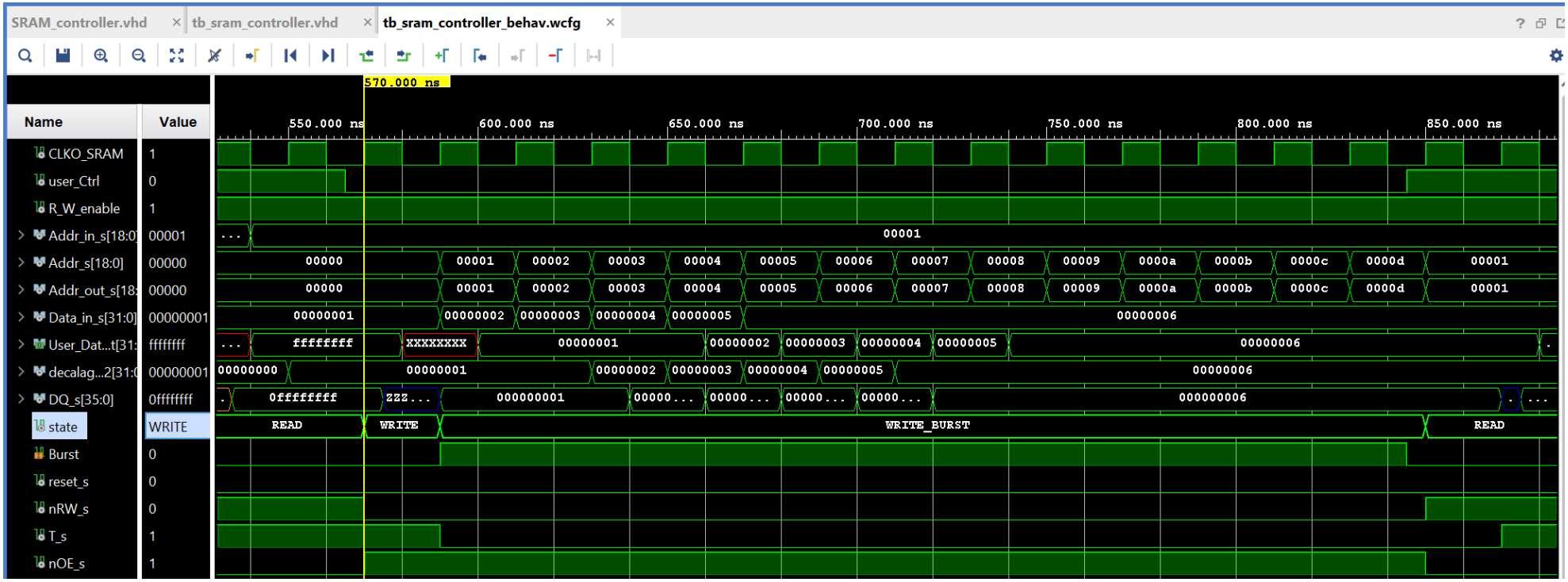


Figure 1.9: Simulation write mode burst

Ensuite nous lisons les données en mode burst en reprenant les mêmes address que pour l'écriture.

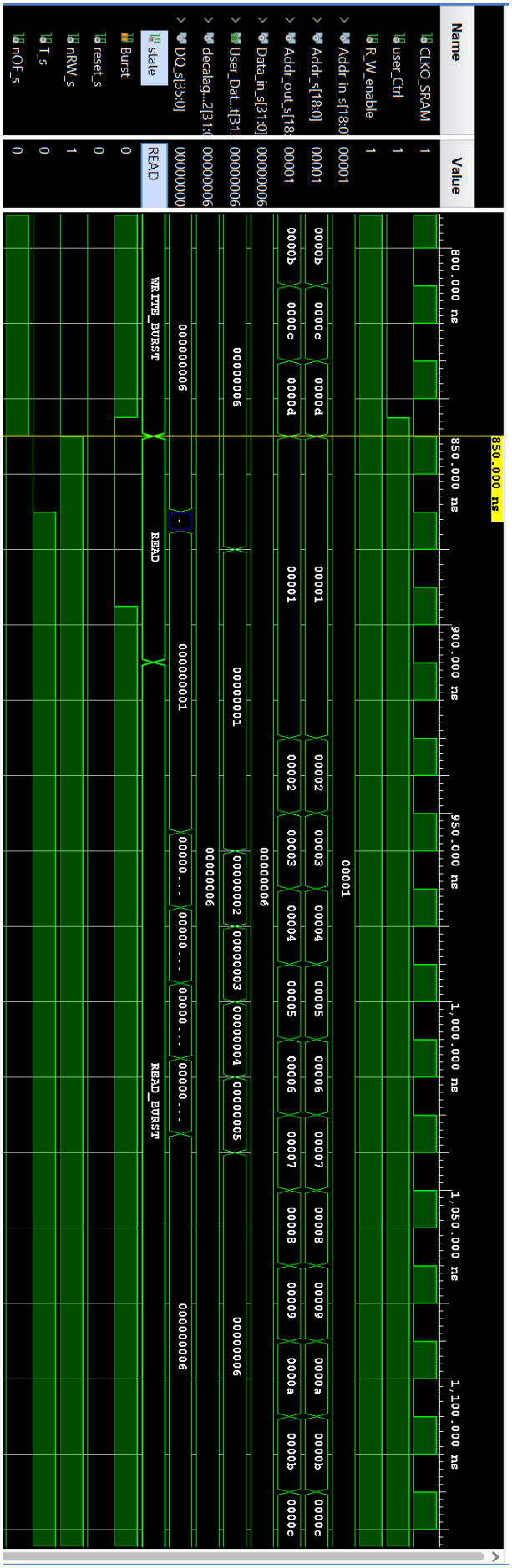


Figure 1.10: Simulation read mode burst

Le fait qu'on ait utilisé un compteur forcé au lieu du Ld_n permet de ecrire/lire plus

de 4 données à la suite car on est limité avec le compteur de la sram.

à partir de là, nous pouvons considérer que le mode burst est fonctionnel.
Il reste cependant quelques améliorations à apporter pour rendre le contrôleur plus robuste et efficace.
Par exemple, l'utilisation de l'incrementation automatique de l'adresse par la SRAM elle-même en mode burst, plutôt que de gérer cela manuellement dans le contrôleur.
Cela nécessiterait une révision de la gestion du signal Ld_n et une adaptation de la logique de la FSM pour s'aligner avec le comportement de la SRAM en mode burst.

1.5 Estimation de la durée d'exécution du contrôleur SRAM

L'estimation de la durée d'exécution du contrôleur SRAM a été réalisée à partir des chronogrammes issus des simulations ModelSim présentées dans ce compte rendu. Ces simulations permettent d'observer précisément l'évolution des signaux en fonction du temps et des cycles d'horloge, et constituent donc une base fiable pour une estimation temporelle du fonctionnement du contrôleur.
Le contrôleur SRAM est entièrement synchronisé sur l'horloge Clk. Chaque transition d'état de la machine à états finis ainsi que chaque opération de lecture ou d'écriture s'effectue sur des fronts d'horloge bien définis. La durée d'exécution d'une opération dépend donc directement du nombre de cycles d'horloge nécessaires à son déroulement.

1.5.1 Accès en mode lecture et écriture simple

D'après les chronogrammes de simulation, une opération d'écriture simple se déroule selon les étapes suivantes :

- Un cycle pour la prise en compte de la commande et de l'adresse utilisateur.
- Un cycles de décalage de la donnée afin de respecter les contraintes temporelles de la SRAM.
- un cycle effectif d'écriture dans la mémoire.

On observe ainsi qu'une écriture simple nécessite environ 3 cycles d'horloge entre la demande utilisateur et la fin effective de l'opération.

De manière similaire, une opération de lecture simple comprend :

- Un cycle pour la validation de la commande et de l'adresse.
- Un cycle pour l'accès mémoire.
- Un cycle pour la récupération et la stabilisation de la donnée en sortie.

La durée d'une lecture simple est donc également de l'ordre de 3 cycles d'horloge, comme confirmé par les marqueurs visibles sur les chronogrammes de simulation.

1.5.2 Accès en mode Burst

En mode Burst, l'estimation est plus favorable. Les captures de simulation montrent que:

Le premier accès du burst présente un coût similaire à une lecture ou écriture simple. Les accès suivants sont réalisés à raison d'un mot par cycle d'horloge.
Ainsi, pour un burst de N mots, la durée totale peut être estimée à:
Un coût initial de quelques cycles (initialisation du burst), puis N cycles supplémentaires pour les N transferts consécutifs.
Les chronogrammes de simulation en mode Burst confirment ce comportement, avec une incrémentation de la donnée et une lecture/écriture valide à chaque cycle d'horloge pendant la phase de burst.

1.5.3 Limites de l'estimation

Cette estimation repose sur un modèle simulé et ne prend pas en compte les délais physiques liés au FPGA, les temps de propagation réels, ni les éventuelles contraintes liées à l'implémentation matérielle finale.

Cependant, dans le cadre de ce projet, l'analyse temporelle basée sur les captures de simulation est suffisante pour caractériser le comportement du contrôleur SRAM et comparer les performances entre les modes simple et burst.

1.6 Conclusion

Ce projet avait pour objectif de concevoir et implémenter en VHDL un contrôleur de mémoire SRAM en respectant les contraintes temporelles et fonctionnelles du composant mt55l512y36f.
Dans un premier temps, l'étude et la validation de l'IO Buffer ont permis de comprendre le fonctionnement du bus bidirectionnel et de garantir une gestion correcte des phases de lecture et d'écriture, notamment grâce à la mise en haute impédance du bus lors des lectures.

La conception du contrôleur SRAM s'est appuyée sur une machine à états finis assurant une gestion fiable des accès mémoire. Les décalages d'horloge entre données, adresses et signaux de contrôle ont été correctement pris en compte à l'aide de registres synchronisés, ce qui a permis de respecter les exigences de la SRAM. Les simulations réalisées à l'aide des test bench ont validé le bon fonctionnement des opérations de lecture et d'écriture sur l'ensemble de l'espace mémoire.

L'implémentation du mode Burst a permis d'améliorer les performances du contrôleur en autorisant des accès consécutifs à la mémoire. Bien que l'incrémentation de l'adresse ait été gérée manuellement par le contrôleur pour des raisons de temps, les simulations montrent que le mode Burst fonctionne correctement en lecture comme en écriture.

Ce projet a ainsi permis de mettre en pratique la conception d'un système matériel complet en VHDL, depuis l'analyse du composant jusqu'à la validation par simulation. Des améliorations restent possibles, notamment en exploitant pleinement les mécanismes internes de la SRAM pour optimiser le mode Burst, mais le contrôleur développé répond

aux objectifs fixés et constitue une base solide pour des évolutions futures.